LLNL-CONF-544131

Performance Portability for Unstructured Mesh Physics

J. A. Keasler

April 2, 2012

DOE Exascale Research Conference
Portland, OR, United States
April 16, 2012 through April 18, 2012

Performance Portability for Unstructured Mesh Physics

Jeff Keasler
Lawrence Livermore National Laboratory
7000 East Avenue
Livermore, California
{keasler}@llnl.gov

ABSTRACT

ASC legacy software must be ported to emerging hardware architectures. This paper notes that many programming models used by DOE applications are similar, and suggests that constructing a common terminology across these models could reveal a performance portable programming model. The paper then highlights how the LULESH mini-app is used to explore new programming models with outside solution providers. Finally, we suggest better tools to identify parallelism in software, and give suggestions for enhancing the co-design process with vendors.

1. INTRODUCTION

Nearly two decades ago, the High Performance Computing community settled on a model of parallel computing based upon commodity hardware (clusters of nodes using mass produced microprocessors) and commodity software (Unix-like Operating Systems with MPI and OpenMP parallel software interfaces). ASC invested hundreds of man years into writing software for this model. We are exploring ways to evolve this legacy programming model to adapt to emerging hardware, or worst case, to try to identify a new model that will be more portable as we face the prospect of new and disruptive hardware technology.

2. PROGRAMMING MODELS

We have observed that many of the largest mesh based physics applications in the DOE have settled on a similar underlying software model to manage their parallel data at scale, but each model is described using different terminology. For example, we have noted that several DOE programming models are based on the concepts of locality contexts, index sets, and entity relations:

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-PROC-xxxxxx).

Locality context Simulation data is defined over mesh subsets. Each subset is often associated with one or more arrays, each array having a length equal to the cardinality of the subset. Examples of locality contexts used in applications include: View(ALE3D), PatchData(SAMRAI), Bucket(SierraToolKit), LayoutData (Chombo), GridFunction(Cactus), Store(Loci), Entity Set-Tags (ITAPS), etc. Mesh subsets can often be arranged in arbitrarily deep hierarchies in these models.

Index set Index sets are used to identify entities within a locality context, or to provide a mapping between entities in child and parent locality contexts. Examples of index sets used in applications include: IndexSet(ALE3D), Box(SAMRAI), Part(SierraToolKit), BoxLayout(Chombo), Grid(Cactus), Map(Loci), EntitySet(ITAPS), etc.

Entity relation Entity relations are used to describe interactions between entities that have been defined on distinct subsets or topological centerings (i.e. node, elements, faces, etc.). Entity relations can also be used to define stencil relations within a single locality context.

By taking a step back and creating a common terminology for concepts found in disparate programming models, we as a community could potentially identify a unifying API that could provide a reusable performance portable programming model to map software to hardware at scale.

3. UNSTRUCTURED MESH MODEL

At Livermore, we have been discussing a programming model with the following features:

Locality context layer This layer is responsible for the declaration, partitioning, placement, ordering, interleave, alignment, and allocation of array data. Each listed responsibility must be implemented to achieve performance portability across diverse architectures.

Index set layer This layer is responsible for decoupling the ordered or unordered traversal of locality contexts from the specific algorithmic kernels being applied. This differs from current programming models where loop traversal is directly bound to the loop body in a way that enforces a specific serialized sequential traversal.

Communication layer This layer is responsible for communicating data between locality contexts. This layer includes a dependency scheduling and flow control component.

In mesh based physics software, we have observed that most loop constructs are dedicated to task based or locality-context based traversals. Examples of locality-context based traversals include (1) a triply nested loop iterating over a 3D block-structured mesh, or (2) a single unstructured mesh loop iterating over entities in arrays referenced directly or indirectly through the loop control variable. We would like to have a single, unified, and performant index set abstraction embodying both traversals (1) and (2).

A block-structured traversal can be represented by (2*number_of_dimensions + 1) integers, which is a compact and low memory bandwidth traversal space representation. Programmers often arrange loop nests based on these integers to achieve highly optimizable stride-one data access. If the compiler can see the stride-one loop nesting, it enables vectorization. Furthermore, this stride-one access pattern can activate hardware prefetch streams making memory movement even more efficient for this kind of abstraction. The traversal pattern for this kind of index set is very specific, and mostly inflexible.

An unstructured traversal is typically implemented by an explicit list of integer indices used to access entries in associated arrays. This list of integers has to be streamed through memory and can consume limited cache resources. Compilers can't typically vectorize unstructured index traversals because the indices can be modified at runtime. Hardware prefetch streams can still work as long as the indices are stride-one ordered. Hardware threads and Instruction Level Parallelism (ILP) can sometimes hide some of the performance issues associated with unstructured index sets. The value of unstructured index traversals is their unlimited flexibility, and their amenability to reordering at runtime.

A hybrid index traversal is described by a run-length-encoded array of structured ranges and unstructured lists. An algorithmic loop body can be bound to a hybrid index set via a C++ lambda function or a C++ functor in such a way that the loop can be versioned into two paths – one for the structured ranges and one for the unstructured lists. The structured range paths can be vectorized at compile time. Furthermore, alignment assumptions can be embedded into the hybrid index set, guaranteeing that structured ranges begin on a SIMD or GPU warp boundary and have a SIMD or warp length granularity. In practice, hybrid index sets consist mostly of sorted stride-one ranges, so the hybrid index set can often be stored with a very compact memory representation, and operations can occur on long vector lengths. Finally, hybrid index sets are amenable to heterogeneous architectures where the unstructured lists are run on the scalar resources and the structured ranges are run on the vector resources of the hardware architecture (possibly simultaneously).

We have implemented hybrid index sets in the context of the Intel C++ compiler, and have found the generated assembly code to be optimally compact and efficient. This proves that the Intel compiler possesses the internal machinery necessary to properly optimize this construct. On the other hand, we have found that the programmers must go to great lengths to make sure these optimizations are applied. We believe that through closer interaction with the compiler vendors, we could suggest simpler ways to expose these optimizations at the source level in a way that will be portable across compilers. Furthermore, such direct compiler vendor interaction could actually result in a more optimized implementation of programming models such as Threading Building Blocks within the Intel compiler. We believe it is the lack of direct contact and working together that prevents the more general availability of these optimizations – not technical barriers.

4. MINI-APP: LULESH

In practice, large legacy software applications are difficult to modify in-place, complicating our ability to explore new programming models and collaborate with vendors. To address this, the DOE community has been creating small test bed applications, or mini-apps, that are designed to be representative of computational patterns found in larger applications. A mini-app must contain sufficient complexity to clearly expose interactions found in real applications if it is to be useful for co-design. On the other hand, a mini-app must be compact enough to allow developers to explore new programming models with only a few hours or days of effort.

The DARPA UHPC program funded work on the Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH) mini-app which was specifically designed to evaluate new architectures and compilers. LULESH consists of a serial reference implementation and thorough documentation, helping system hardware and software developers to understand key aspects of unstructured mesh based physics.

LULESH has since been extended to work with OpenMP, CUDA, MPI, Chapel (HPCS), Loci (parallel functional/relational model), and a few other software systems. Several algorithmic variants of LULESH have been created to explore software maintenance vs. performance tradeoffs. A preliminary fault tolerant version of LULESH has been explored by Maya Gokhale at LLNL. Academic collaborators at Rice University and Ohio State University have worked with LULESH. The Sandia SST simulator has been applied to LULESH and a paper on that work is in submission to SC12. LULESH is a key application being used by the AATEMPS, AASD, ExMatEx, and "Data Abstractions for HPC" projects at LLNL to explore scalability on future hardware systems.

5. CASE STUDIES: CHAPEL AND LOCI

LULESH has been ported to the Chapel and Loci programming models. Chapel is designed to answer the question, "How can I decompose parallel performance portability into manageable subtasks?" while Loci answers the question, "How can I assemble manageable subtasks into a parallel and performance portable program?" In other words, Chapel takes a top-down approach to parallelism, while Loci takes a bottom-up approach.

Despite different approaches to parallelism, Chapel and Loci are very similar. Both models strive to make the parallel program look like a serial program to help simplify reasoning about parallelism and to improve software maintenance.

Both models reduce the size of parallel programs when compared to traditional parallel programming techniques. Both models have sophisticated features to help tailor software to a diversity of hardware environments with little additional work from the programmer. Both models rely on index sets as a key abstraction to simplify software maintenance and attain portable performance.

The Chapel programming model uses index sets through the Domain concept, which provides a global namespace to label data entities used within parallel programs. Chapel allows programmers to tune Chapel applications to different hardware environments using DomainMaps and other sophisticated object-oriented features. The Chapel team made an initial port of LULESH to Chapel that used a block-structured mesh abstraction. While the block-structured port was true to the implementation of the LULESH mini-app, it was clear to the Chapel team that perhaps this port was not in the spirit of what the LULESH mini-app really meant to capture. This initial Chapel port of LULESH led to additional collaboration between the LLNL and Chapel teams. A face-to-face multi-day meeting was arranged where we were able to port the block-structured Chapel LULESH to be a completely unstructured mesh application in three to four hours with very few changes to the original Chapel source code. Some additional hours were needed to debug unstructured mesh boundary conditions. As the result of our direct interaction over several days, we exchanged domain-specific knowledge that lead to discussions about improvements to the LULESH software structure and possible optimizations that could be added to the Chapel compiler to improve Chapel's performance on unstructured (and possibly block-structured) mesh physics applications. This direct pair-programming interaction between compiler vendor and application developer was vital to understanding how to make needed changes to both the compiler and the application.

The Loci programming model uses index sets through the Map concept, which implements a relational data model between mesh entities. Functions in Loci are specified in terms of rules applied to data entities. The dependencies implied by these rules are assembled to form a unique execution schedule that is logically consistent. The assembly of the schedule is flexible enough to map the functions in the program to the functional units of the underlying hardware. The Loci model was created at Mississippi State University, in part, from knowledge of unstructured mesh idioms gleaned from working directly as a developer on DOE ASC software applications. Loci is now used by NASA's Marshall Space Flight Center for its ability to automatically parallelize serial source code, and to exploit the logical consistency features (and hence correctness checking) not found in other languages. Again, direct knowledge of the application space is what made this full featured programming model possible.

6. METRICS

We have been struggling to develop a set of meaningful performance metrics that can be used to help characterize the scalability of our software. The best publically available tools we have found have been Intel's Parallel Advisor and the Embla data dependence profiling tool. We believe there is a glaring absence of a tool that provides an available parallelism metric that would identify sections of applications that have dependencies that effectively serialize associated algorithms. We believe that compilers store a lot of this dependency information internally, and that if compilers could simply output statistics on the static dependence distances they have found, it could make it much easier to evaluate scalability. Adding a dynamic dependence distance evaluation tool would give us an even better picture of the resource bottlenecks in our large applications.

7. LOOKING FORWARD

A cost-effective and results-oriented way to move scalable programming forward is through direct interaction between the application developers and the vendor compiler and runtime developers. One of the strengths of Lawrence Livermore National Laboratory has been the way we co-locate team members with diverse backgrounds in adjacent offices. In the past this has allowed Livermore to produce high quality integrated software products in terms of both performance and capability. While other institutions have had great success at deep dives in basic research areas, LLNL believes that integrating available capabilities across disciplines into a functional system level product is the key to true success. Livermore has found that this kind of success requires direct and co-located interaction.

A second approach to encourage scalable performance would be to produce a multi-lab suite of core kernels akin to what the Livermore Loops benchmark provided in the past. Current application developers at each of the national laboratories have encountered software constructs that should be optimizable, given the compilers ability to analyze the constructs being used. We have often found, for each construct, say, two out of three vendor compilers will optimize the construct correctly, but one will not. This sounds like a good situation until one realizes that applications have dozens of constructs that need to be optimized simultaneously, and a two in three chance of optimization for each construct has a multiplicative effect that results in abysmal performance using any given compiler. By providing a test suite along with tests for expected performance behavior, we could write RFPs for new hardware to include this test suite. This could encourage compiler support for scientific software constructs that have long been needed, or at minimum encourage more interaction between app and compiler people.

8. SUMMARY

NNSA has many large legacy applications that we need to port to emerging architectures with minimal effort. Exploring multiple portability options directly in large software application is impractical. To solve this problem, we create proxy applications with sufficient complexity to be representative of complex data interactions found in real software. Proxy applications must also be compact enough to act as experimental test beds for a variety of potential future programming models.

We believe there are many similar constructs being used across most large science applications, but there is no common terminology to describe the high level concepts shared by these applications. By fostering a common terminology, we believe that great progress could be made in discovering a programming model that would work well across mesh based physics software on future architectures.

Through our exploration of the LULESH mini-app, we have found that direct vendor interaction has been (or would be) the most profitable way to solve many of the programming model issues we have found. Directly interacting with vendors to address existing low level optimization deficiencies could lead researches to create better programming models using less effort. Barring direct interaction, we might at least produce a representative tri-lab suite of application kernels that we believe should be optimizable by existing compilers, but are not.

We would greatly benefit from a new tool that yields an available parallelism metric to pinpoint algorithms that need to be rewritten, or to measure how scalable our existing applications can be, either in theory or in practice. Additionally, if we had better compilers as described above, existing performance metrics would likely become more meaningful due to eliminating the noise caused by unpredictable and variable optimization quality in current compilers.

Finally, we believe direct collaboration between app developers and compiler developers in the context of mini-apps or larger legacy applications is vital. From our experience with LULESH, we have found that the application developers can only transfer domain-specific knowledge to compiler writers (and vice-versa) by exposing compiler behavior on application kernels in an interactive co-located working environment.